SESSION 9

iOS DEVELOPMENT

HOW TO WRITE

CLEAN CODE

WHAT IS CLEAN CODE?

- Clean code is code that is easy for humans to understand
    - It follows consistent, predictable patterns
    - It’s organized in a logical way
    - It provides the reader context about the intended functionality

WHY IS CLEAN CODE IMPORTANT?

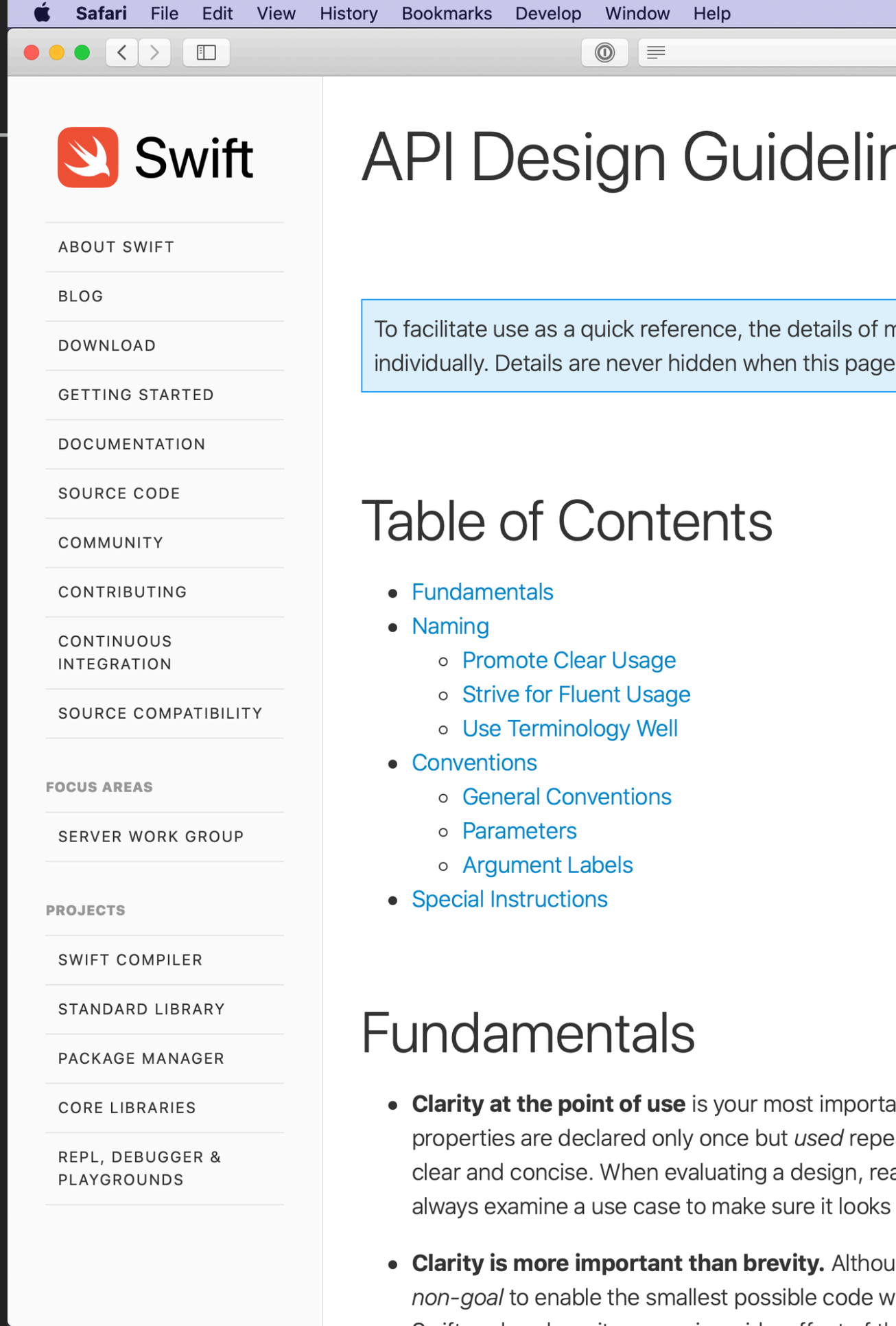
- Clean code is easier to maintain.
    - New features can be added more quickly
    - Bugs are less likely to be introduced

WHO DEFINES CLEAN CODE?

- “Clean code” is somewhat subjective. However, there are certain guidelines that are widely accepted.

WHO DEFINES CLEAN CODE?

- Start with the official Swift API Design Guidelines

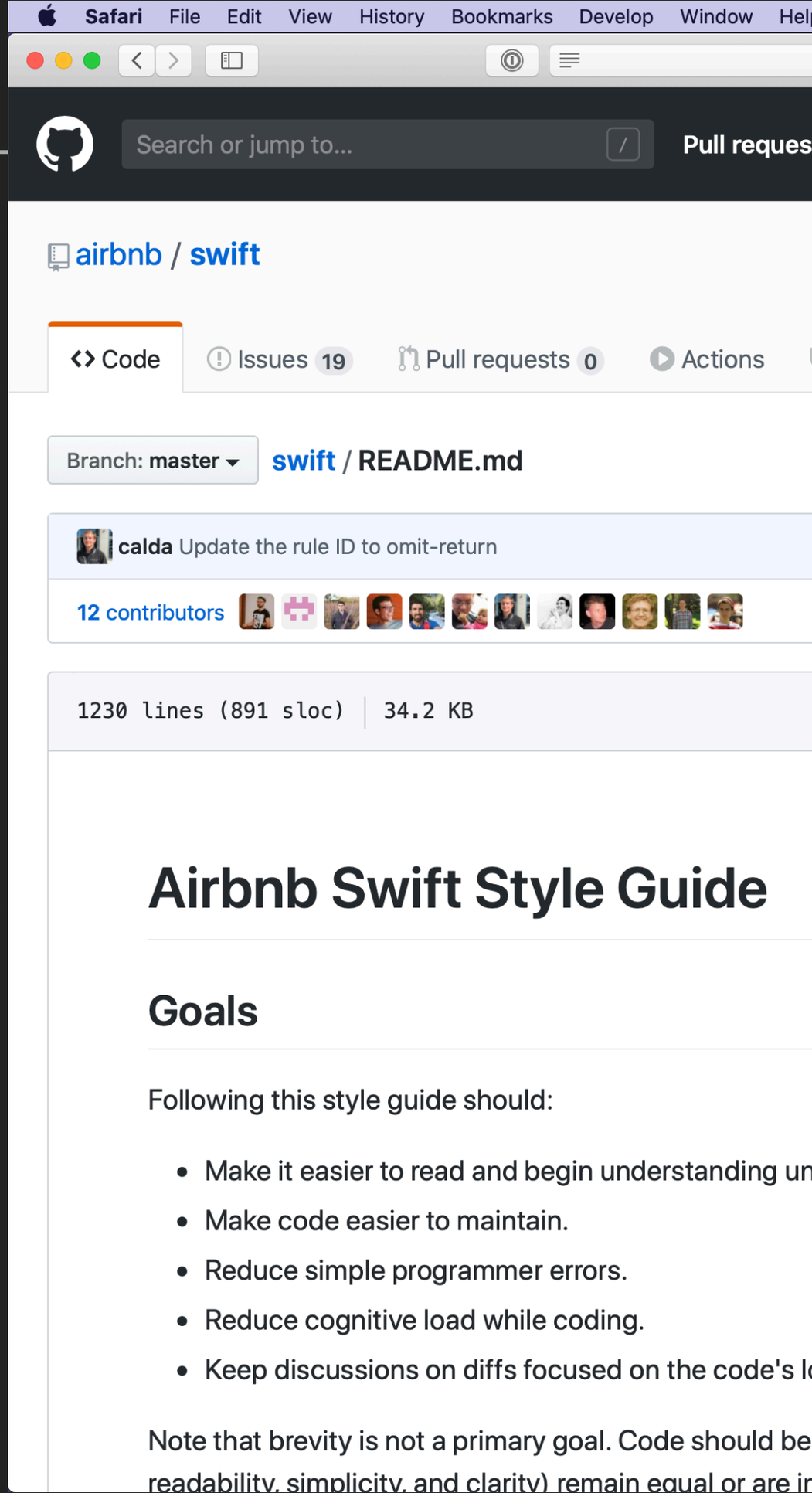


WHO DEFINES CLEAN CODE?

- Take your cues from Apple
    - If you’re unsure about how something should be written, try to find a similar example in one of Apple’s frameworks

WHO DEFINES CLEAN CODE?

- Look at the Swift style guides published online
- Companies like Google, LinkedIn and Airbnb have publicly available style guides



WHO DEFINES CLEAN CODE?

- Discuss code standards with your team
    - Talk about naming conventions, formatting, etc.
    - Document developer agreements
- Allow your agreements to evolve over time

CLEAN CODE IN SWIFT

- We'll go over some best practices for writing clean code in Swift
    - Everything we discuss is widely accepted by the Swift community
- Follow these practices in your final projects

CLEAN CODE

ORGANIZATION

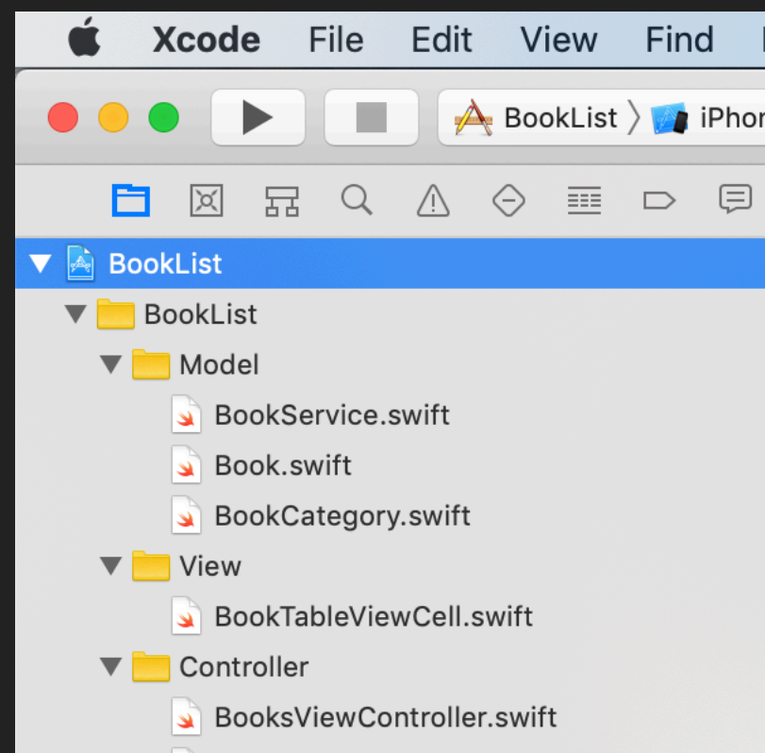
ORGANIZATION

- Rule #1: Put every class, struct and enum into its own file
    - Each custom type should live in a file with the same name
    - This makes your project easier to navigate

Note: Some developers will break this rule for very small structs or enums

ORGANIZATION

- Rule #2: Organize files into groups (folders)
  - You can group by feature: Settings, Login, Home
  - Or you can group by role: Model, View, Controller



ORGANIZATION

- Rule #3: Use consistent spacing and indentation
    - Make sure curly braces line up properly
    - Use `Control+I` to format a selected block of code

ORGANIZATION

- Rule #4: List all properties first, then methods
  - Don’t scatter properties throughout a class
  - Instead, group them together at the top

```
class BooksViewController: UIViewController {

    @IBOutlet weak var tableView: UITableView!

    private let bookService = BookService()
    private var books: [Book] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        tableView.dataSource = self
        books = bookService.fetchBooksFromFile()
        tableView.reloadData()
    }
```

ORGANIZATION

- Rule #5: Delete dead code
  - Dead code = code that is never called
  - This includes commented-out code

CLEAN CODE

NAMING

NAMING

- Rule #1: Use camel case (not snake case)
  - ✓ Camel case = `titleLabel`
  - X Snake case = `title_label`

NAMING

- Rule #2: Begin types and protocols with a capital letter
    - Classes, structs, enums and protocols start with a capital letter
    - Everything else (functions, variables, etc.) start with a lowercase letter

NAMING

- Rule #3: Choose names that provide context to the reader
    - Think about what you want to convey when something is used, not just when it’s declared

```
@IBOutlet var movieTitleLabel: UILabel!
```
✓

```
@IBOutlet var movieTitle: UILabel!
```
X

```
@IBOutlet var label1: UILabel!
```
X

NAMING

- Rule #4: Favor clarity over brevity
    - Spell out complete words (avoid abbreviations)

```
func loadBkgdImage() -> UIImage? {
    // ...
    return image
}
```

```
func loadBackgroundImage() -> UIImage? {
    // ...
    return image
}
```

NAMING

- Rule #4: Name boolean properties and methods so that they read as assertions
    - For example:
        - `name.isEmpty`
        - `square.intersects(triangle)`

CLEAN CODE

CONTROL FLOW

CONTROL FLOW

- Rule #1: Use `guard` to avoid nesting

```
@IBAction func unwindToBookList(_ sender: UIStoryboardSegue) {
    if let addBookViewController = sender.source as? AddBookViewController {
        if let book = addBookViewController.book {
            books.append(book)
            bookService.saveToFile(books: books)
            tableView.reloadData()
        }
    }
}
```

X

```
@IBAction func unwindToBookList(_ sender: UIStoryboardSegue) {
    guard let addBookViewController = sender.source as? AddBookViewController else { return }
    guard let book = addBookViewController.book else { return }
    books.append(book)
    bookService.saveToFile(books: books)
    tableView.reloadData()
}
```

✓

CONTROL FLOW

- Rule #2: Don’t explicitly compare booleans to `true` and `false`

```
if isFavorite == true {
    starImageView.isHidden = false
} else {
    starImageView.isHidden = true
}
```

X

```
if isFavorite {
    starImageView.isHidden = false
} else {
    starImageView.isHidden = true
}
```

✓

CONTROL FLOW

- Rule #3: Don’t use an index when you could iterate over the array directly

✗

```
for i in 0..<books.count {
    let book = books[i]
    print(book.title)
}
```

✓

```
for book in books {
    print(book.title)
}
```

CONTROL FLOW

- If you need the index as well as the element, use the `enumerated` method.

```
for (index, book) in books.enumerated() {
    print("\(index) \(book.title)")
}
```

CLEAN CODE

LEVERAGE SWIFT'S SAFETY

LEVERAGE SWIFT'S SAFETY

- Rule #1: Make properties and methods `private` when they are only accessed within the class
    - By default, properties and methods are accessible from anywhere in your project

```
private var somePrivateProperty: String = "shhh..."

private func somePrivateMethod() {
    // ...
}
```

✓

LEVERAGE SWIFT'S SAFETY

- Rule #2: Use `let` instead of `var` whenever possible
    - Only use `var` when a variable is actually modified somewhere in your code

LEVERAGE SWIFT'S SAFETY

- Rule #3: Use non-optionals instead of optionals whenever possible
    - Only use an optional when a variable might actually be nil somewhere in your code

ONE MORE THING...

- Don't use semicolons!

```
let book = books[indexPath.row];
cell.titleLabel.text = book.title;
cell.authorLabel.text = book.author;
```

X

```
let book = books[indexPath.row]
cell.titleLabel.text = book.title
cell.authorLabel.text = book.author
```

✓

ADVICE FOR WRITING CLEAN CODE

- Pay attention to the code you're writing as you go
- Notice established patterns in the codebase
- Review your own pull requests

PUBLISHING YOUR APP TO THE

APP STORE

CHOOSING A BUSINESS MODEL

- Step 1: Choose a business model for your app
  - Free (with or without ads)
  - Freemium
  - Subscription
  - Paid
  - Paymium

CHOOSING A BUSINESS MODEL

- Free app
    - May see more downloads, since users don’t have to pay anything
- Free app with ads
    - Apple discontinued their ad service, iAd, in 2016
    - 3rd-party options, like Google’s AdMob, are still available

CHOOSING A BUSINESS MODEL

- Freemium
  - Downloading the app is free
  - In-app purchases are available for premium features or additional content

In-App Purchases	
Lives Refill	$0.99
Power Up + 5 Moves	$0.99
4 Booster Boxes	$0.99
8 Booster Boxes	$1.99
25 Booster Boxes	$4.99
Power Up: 5 Moves +1 Bomb	$0.99
Power Up: 5 Moves + Fire Extingui...	$0.99
Power Up: 5 Moves + Ice Cracker	$0.99
150 Booster Boxes	$24.99
Week of infinite lives	$4.99

Two Dots
A beautiful puzzle adventure

4.7, 68K Rating | Editors' Choice Apps | #13 Board | 4+ Age

What's New Version History

Version 5.19.3 5d ago

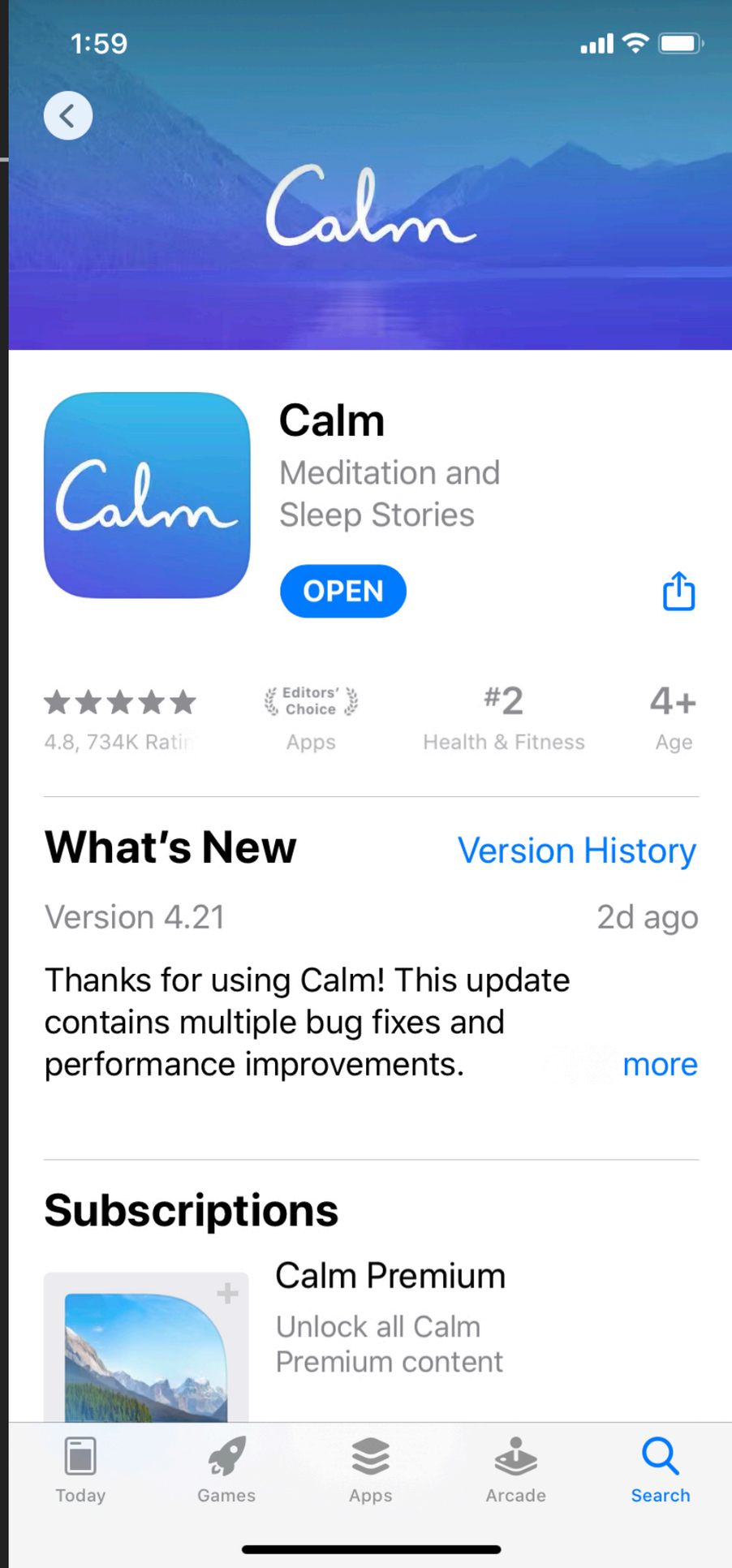
Ongoing improvements to stability and performance.

Preview

Today Games Apps Arcade Search

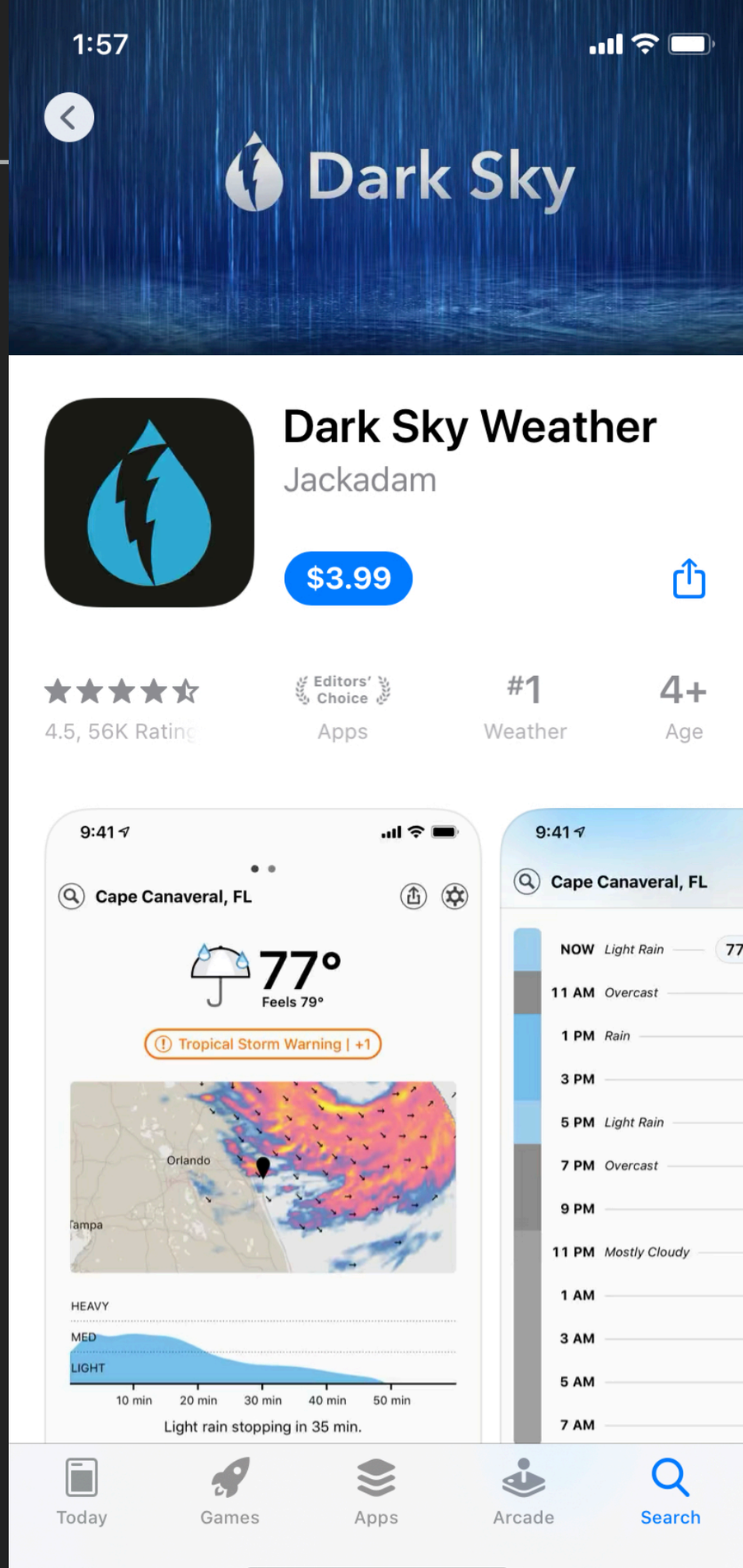
CHOOSING A BUSINESS MODEL

- Subscription
  - Users buy subscriptions through in-app purchases
- This works best if you provide new content on a regular basis



CHOOSING A BUSINESS MODEL

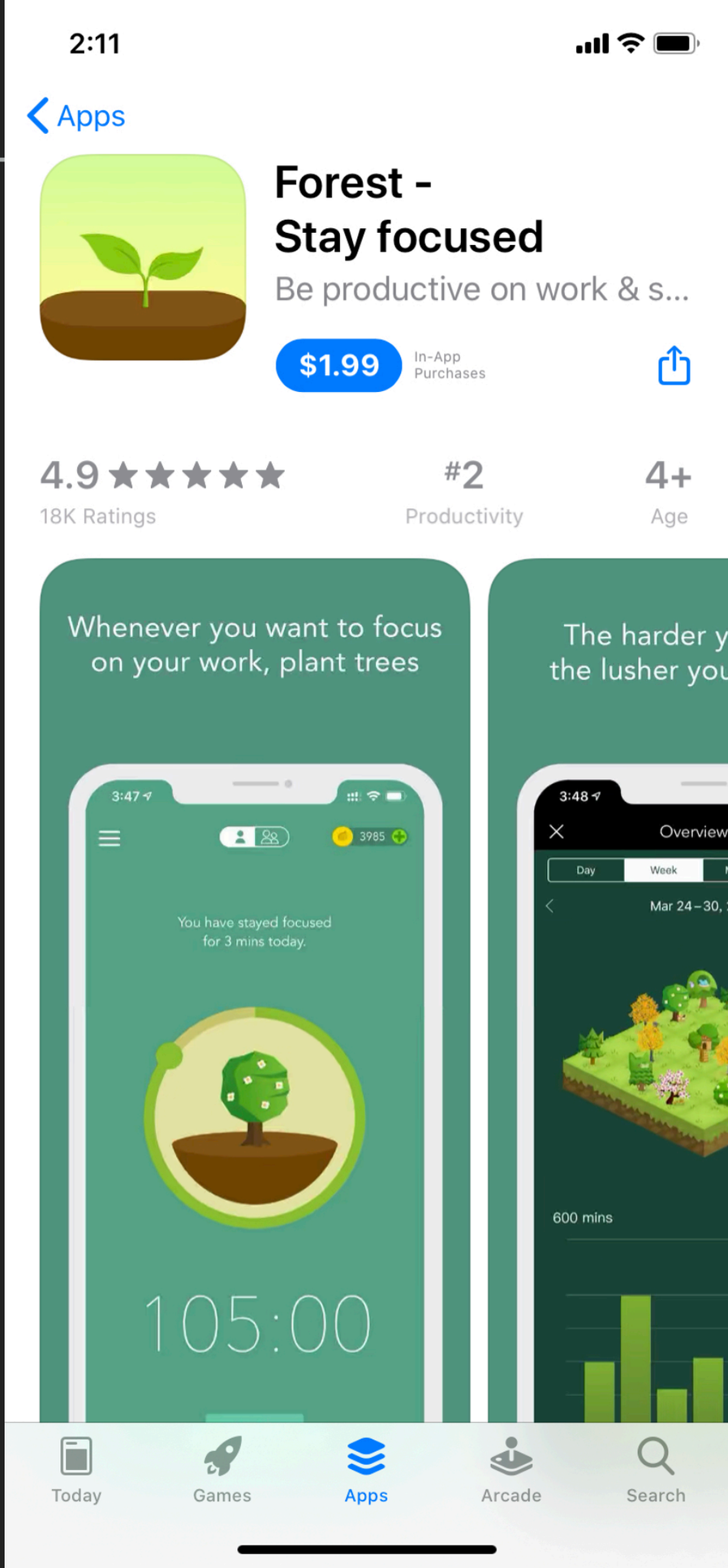
- Paid
  - Users pay to download the app
  - Effective marketing is important, since users don’t get to try out your app before paying



CHOOSING A BUSINESS MODEL

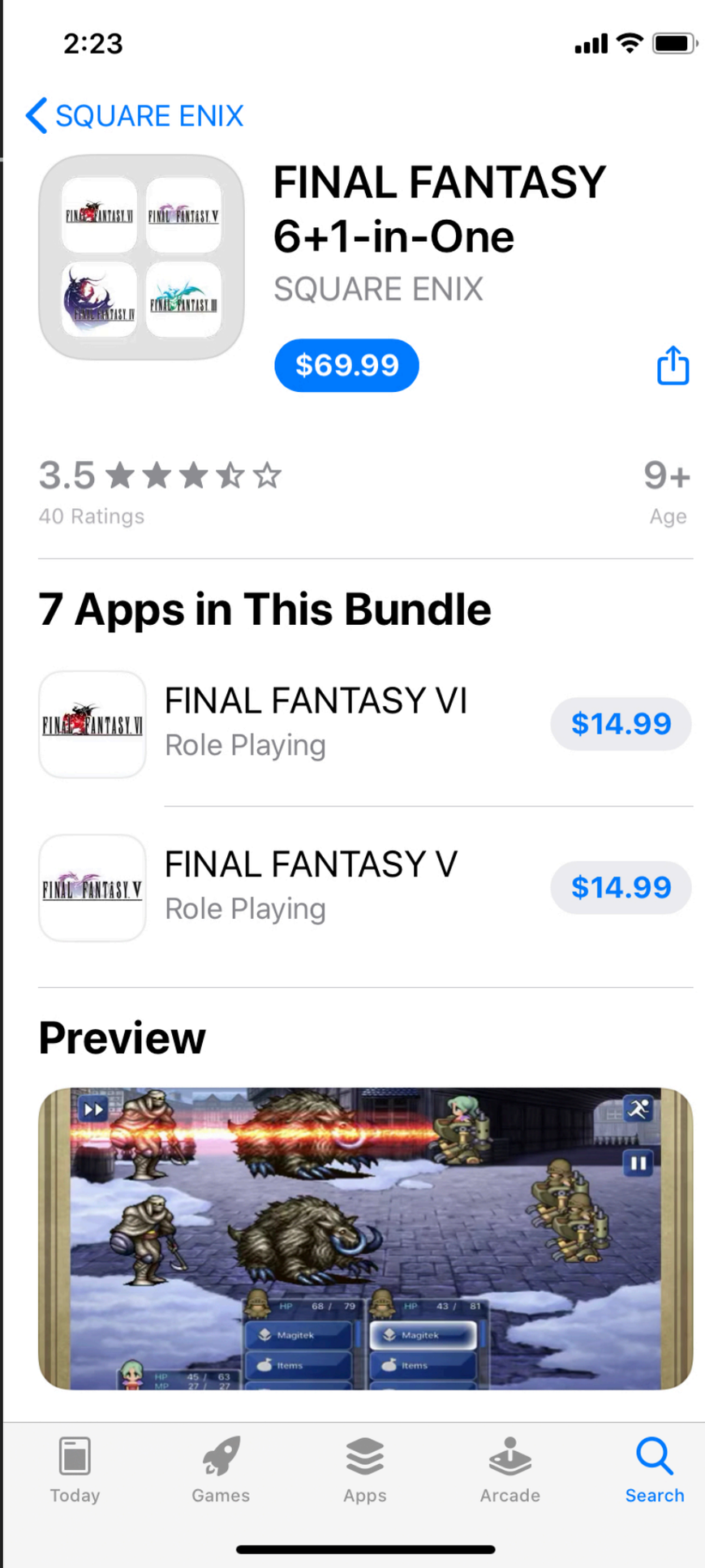
- Paymium
  - Users pay to download the app
  - Additional features are available through in-app purchases
- Set users' expectations carefully!

In-App Purchases	
Bottle of Sunshine Elixir	$0.99
Box of Sunshine Elixir	$1.99
Large bag of Crystal	$9.99
Shovelful of Crystal	$0.99
Small pot of Crystal	$4.99
Cartful of Crystal	$19.99



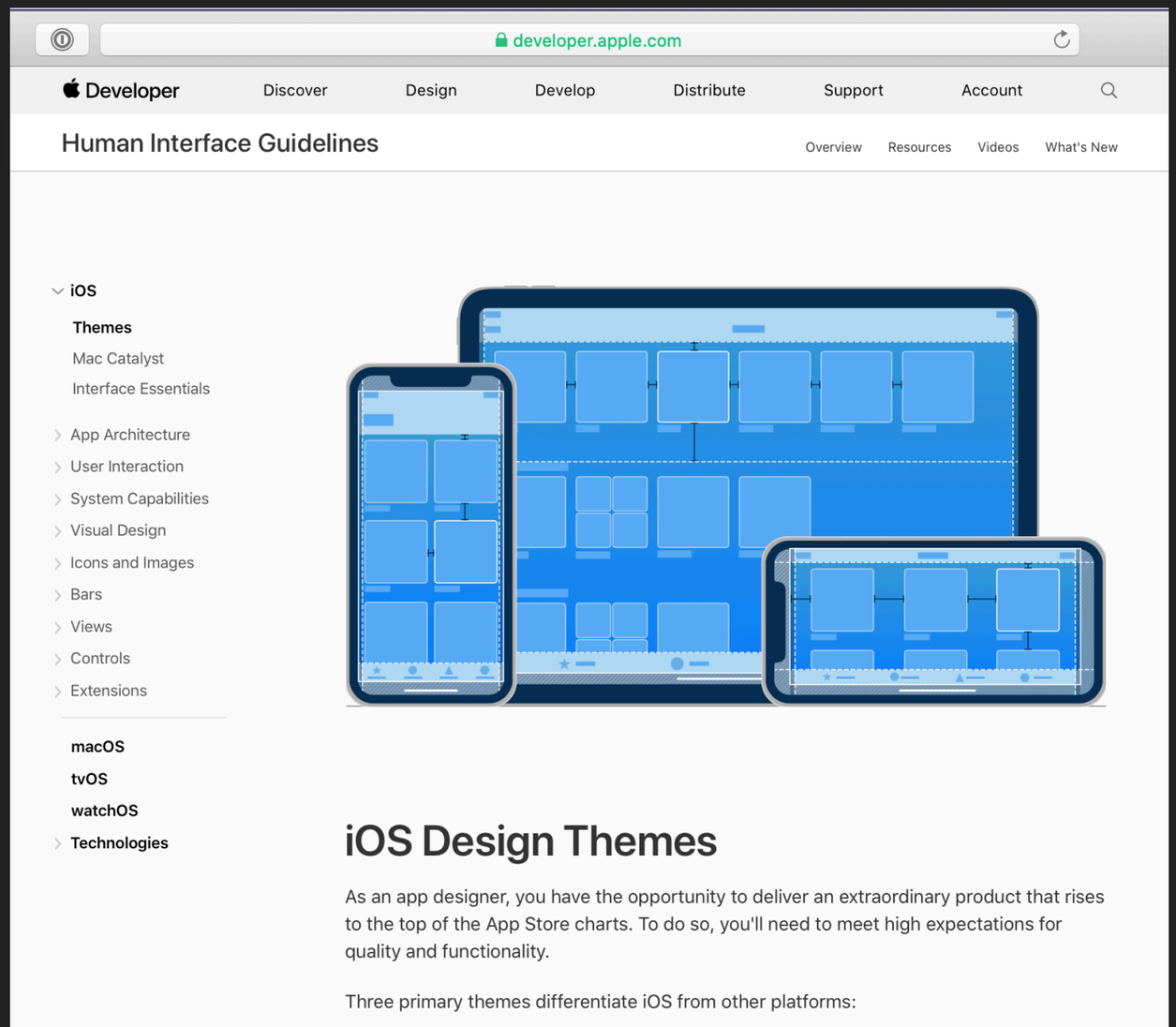
CHOOSING A BUSINESS MODEL

- App Bundles
  - Up to ten apps sold together
  - Users pay less than they would buying each app individually



HUMAN INTERFACE GUIDELINES

- Step 2: Review Apple's Human Interface Guidelines (HIG)



HUMAN INTERFACE GUIDELINES

- The Human Interface Guidelines cover things like:
  - App navigation
  - Gestures
  - Views and controls
  - And a lot more…

APP STORE

HUMAN INTERFACE GUIDELINES

Safari File Edit View History Bookmarks Develop Window Help 31% Mar 3 8:15 PM

developer.apple.com

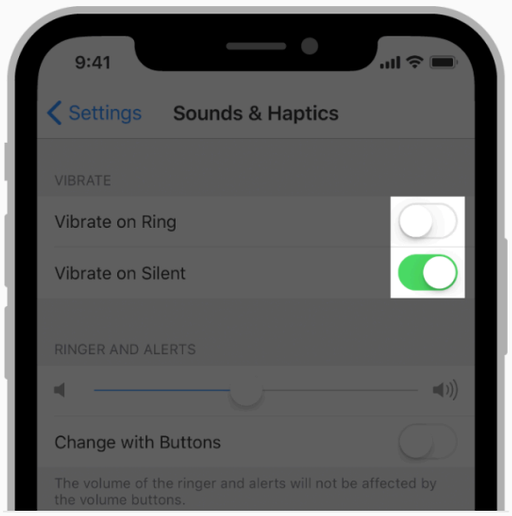
Developer Discover Design Develop Distribute Support Account

Human Interface Guidelines Overview Resources Videos What's New

- iOS
- App Architecture
- User Interaction
- System Capabilities
- Visual Design
- Icons and Images
- Bars
- Views
- **Controls**
  - Buttons
  - Context Menus
  - Edit Menus
  - Labels
  - Page Controls
  - Pickers
  - Progress Indicators
  - Refresh Content Controls
  - Segmented Controls
  - Sliders
  - Steppers
  - **Switches**
  - Text Fields

Switches

A switch is a visual toggle between two mutually exclusive states — on and off.



Consider tinting a switch to match the style of your app. If it works well in your app, you can change the colors of a switch in its on and off states.

Use switches in table rows only. Switches are intended for use in tables, such as in a list of settings that can be toggled on and off. If you need similar functionality in a toolbar or navigation bar, use a button instead, and provide two distinct icons that communicate the states.

HUMAN INTERFACE GUIDELINES

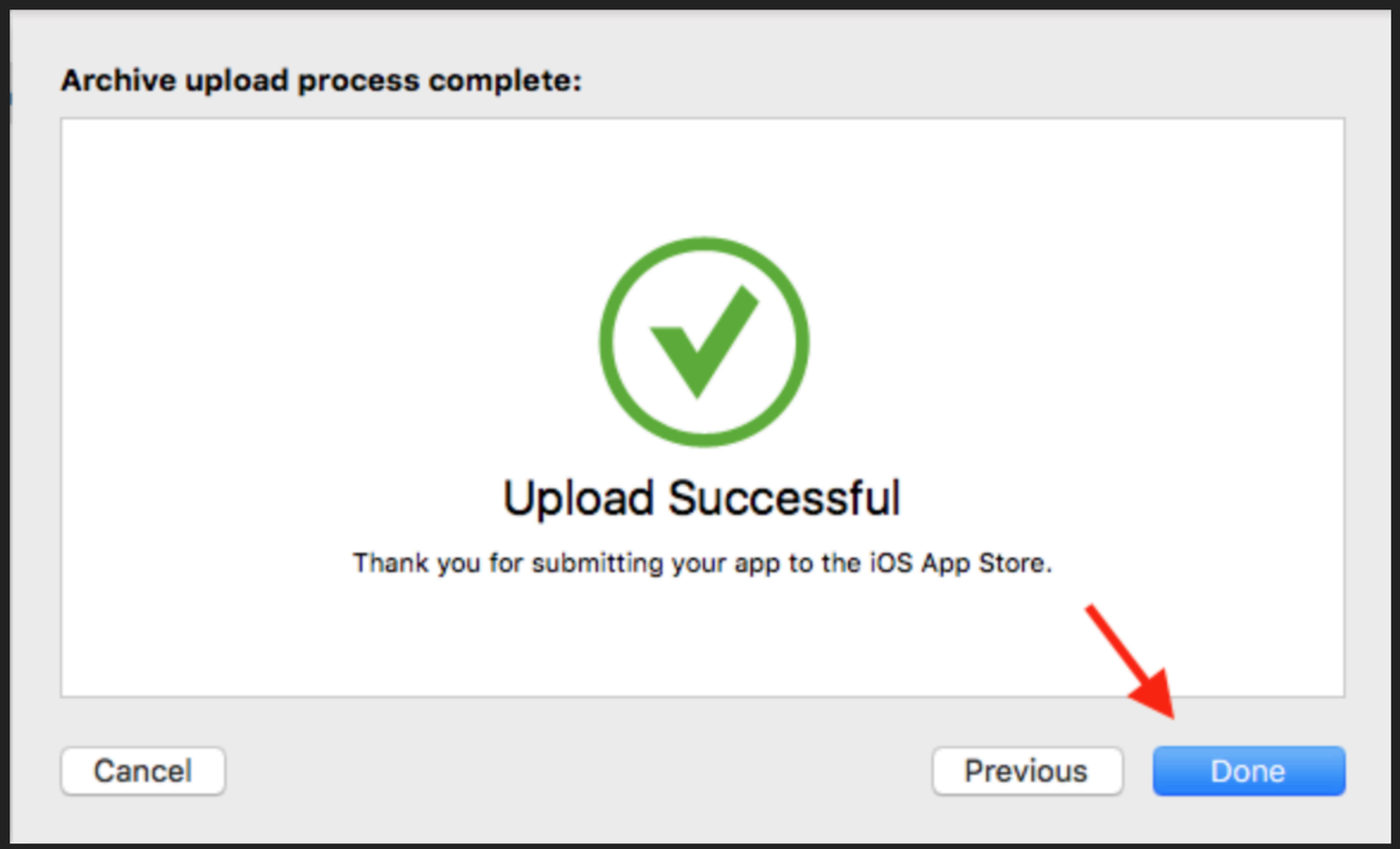
- You don’t have to follow the Human Interface Guidelines exactly (Apple doesn’t!)
- Treat them as guidelines – especially when you aren’t sure how something should work

TEST FLIGHT

- Step 3: Beta test your app with TestFlight
  - Upload your app to App Store Connect
  - Distribute it to beta testers, who can download it through the TestFlight app

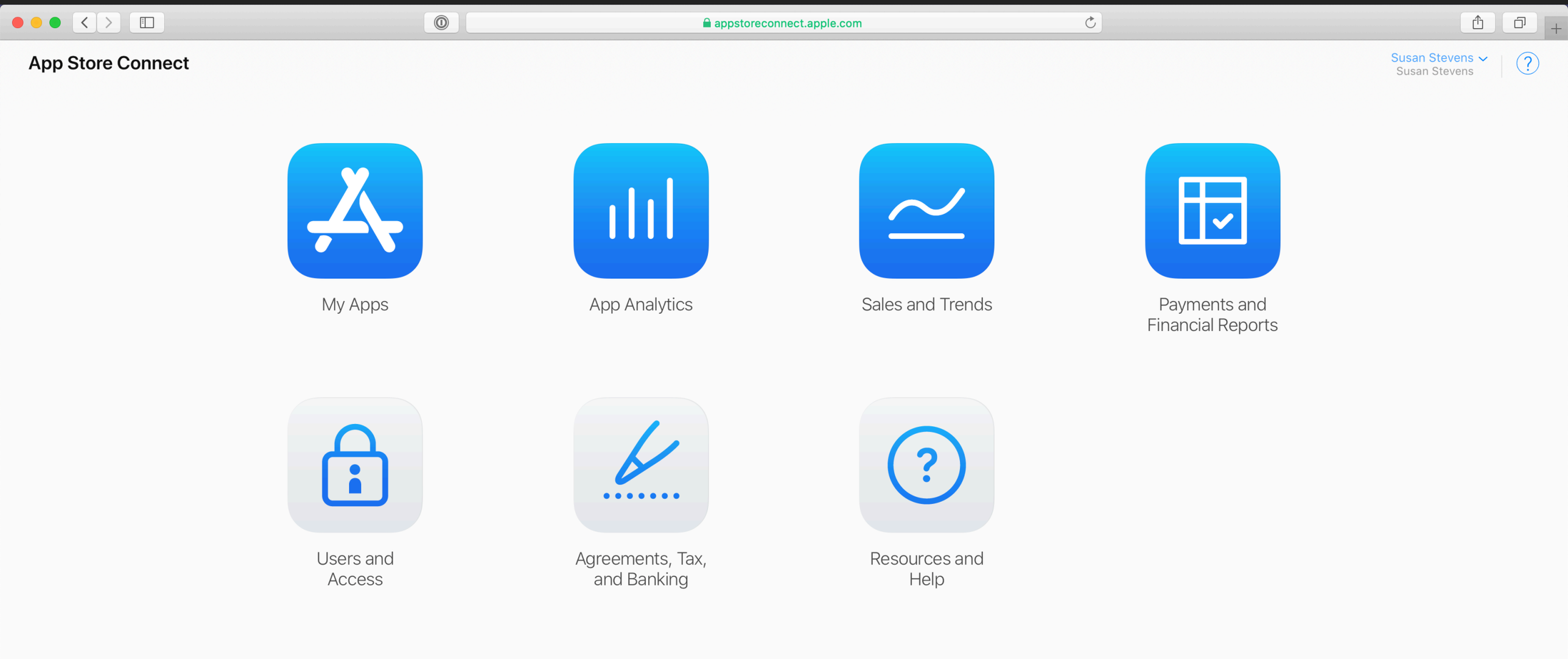
TEST FLIGHT

- Upload and archive your app through Xcode



TEST FLIGHT

- Go to https://appstoreconnect.apple.com

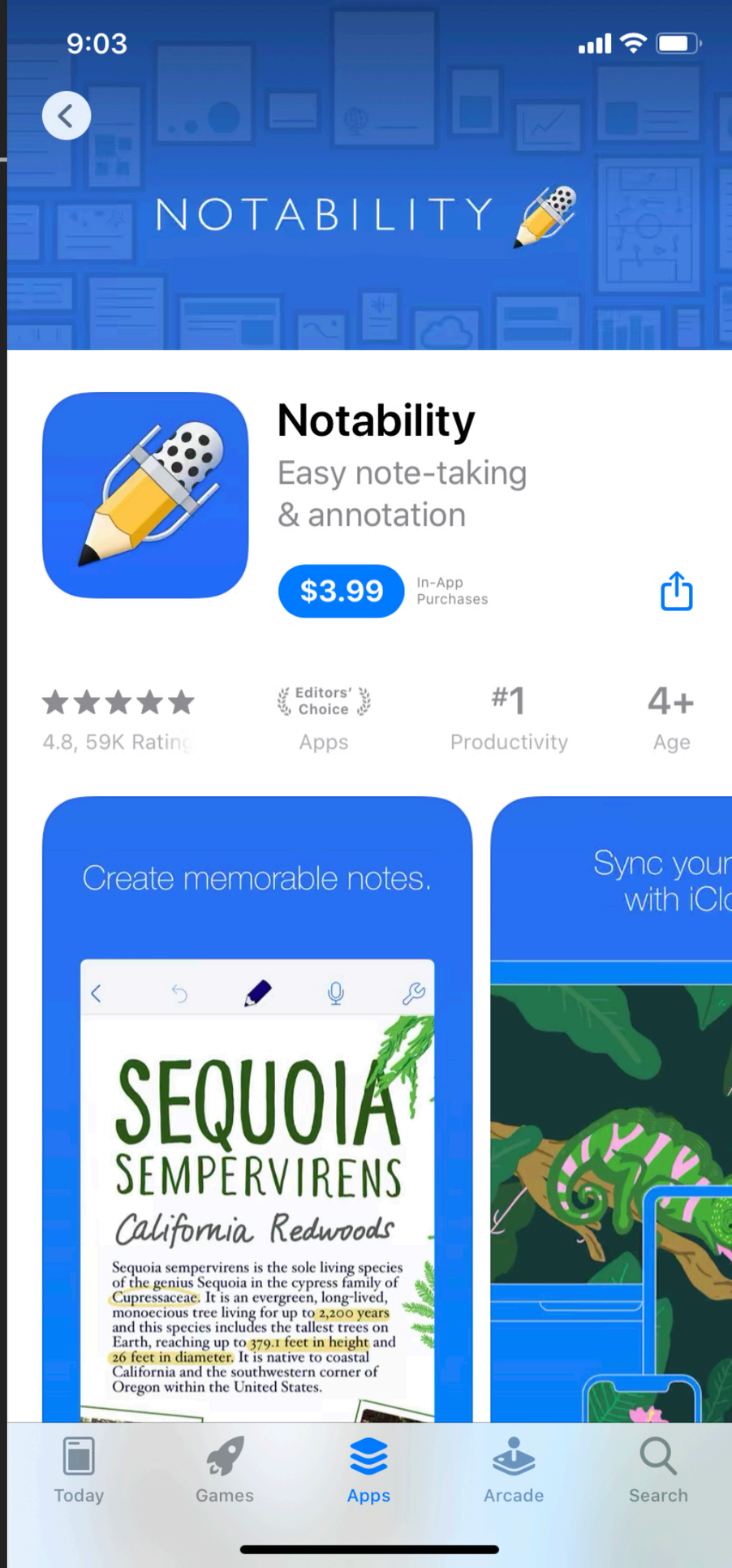


TEST FLIGHT

- From App Store Connect's TestFlight section, you can make your app available to:
  - Up to 25 internal testers
  - Up to 10,000 external testers
- TestFlight builds will work for 90 days

APP STORE PRODUCT PAGE

- Step 4: In App Store Connect, add information for your app's product page
  - App name and subtitle
  - Previews and screenshots
  - Description
  - Category and keywords



PREPARE FOR REVIEW

- Step 5: Prepare to submit your app through App Store Connect
- You'll need to decide:
  - Automatic or manual release once approved?
  - Immediate release to all users or phased release?
  - Reset summary rating?

SUBMIT APP FOR REVIEW

- Step 6: Submit your app for review
    - Apple will review your app to make sure it meets the App Store Review Guidelines
    - Reviews times are typically 24-48 hours
    - You can check the status on App Store Connect

CELEBRATE

- Step 7: 🎉🎉🎉

COMMON APP STORE REJECTIONS

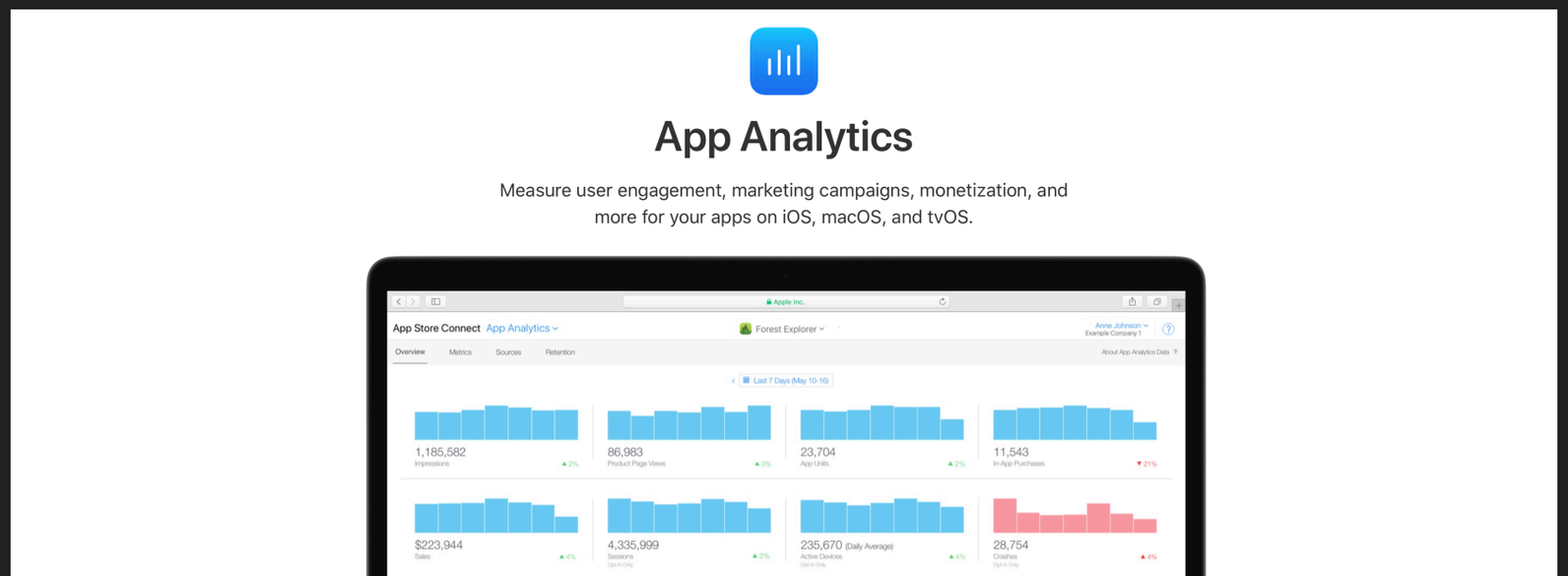
- Apple lists common reasons for rejecting apps, including:
  - Crashes and bugs
  - Inaccurate screenshots
  - Substandard user experience

Inaccurate Screenshots

App Store screenshots should accurately comm
functionality. Use text and overlay images to hi
not obscure it. Make sure app UI and product i
device type in App Store Connect. This helps u
makes for a positive App Store experience.

AFTER LAUNCHING YOUR APP

- After launching your app, you'll be able to:
  - View analytics on App Store Connect
  - Respond to reviews
- Submit updates (each update requires a new review)



BETTER CODE QUALITY WITH

TESTS

MANUAL TESTING

- As you develop an app you are (hopefully) practicing manual testing:
    - Step 1: Write some code
    - Step 2: Build & run the app
    - Step 3: Use the app (see if anything breaks or behaves in unexpected ways)

MANUAL TESTING

- What are some of the disadvantages of this method of testing?

AUTOMATED TESTING

- As an app grows in size and complexity, it becomes harder to manually test all of the ways a user could interact it.
- The solution? Write code to test the app. This is automated testing.

TYPES OF TESTS

- A unit test checks that a small piece of code works as expected in isolation
- An integration test tests the interaction between multiple components
- A UI test checks that the user interface responds as expected

EXAMPLE

```swift
func average(of values: [Double]) -> Double {
    var sum: Double = 0
    for value in values {
        sum += value
    }

    return sum / Double(values.count)
}
```

← Implementation

```swift
func testAverage() {
    let values = [2.5, 3.0, 6.5]
    let avg = average(of: values)
    XCTAssertEqual(avg, 4.0)
}
```

← Unit Test

GIVEN-WHEN-THEN

- A common pattern for structuring tests is given-when-then
    - Given a certain conditions
    - When I call this method
    - Then I get this result
- This is also known as arrange-act-assert

GIVEN-WHEN-THEN

```
func testAverage() {
    // given this array of numbers
    let values = [2.5, 3.0, 6.5]

    // when I call average(of:)
    let avg = average(of: values)

    // then I get 4 as a result
    XCTAssertEqual(avg, 4.0)
}
```

← Given

← When

← Then

MAKING ASSERTIONS

- XCTest is the test framework for Xcode
- It includes a variety of functions for making assertions about your code

MAKING ASSERTIONS

- Assert that an expression is true or false
  - `XCTAssertTrue`
  - `XCTAssertFalse`

MAKING ASSERTIONS

- Assert that an optional is nil or is not nil
  - `XCTAssertNil`
  - `XCTAssertNotNil`

MAKING ASSERTIONS

- Assert that two values are or are not equal
  - `XCTAssertEqual`
  - `XCTAssertNotEqual`

MAKING ASSERTIONS

- Assert that an expression does or does not throw an error
    - `XCTAssertThrowsError`
    - `XCTAssertNoError`

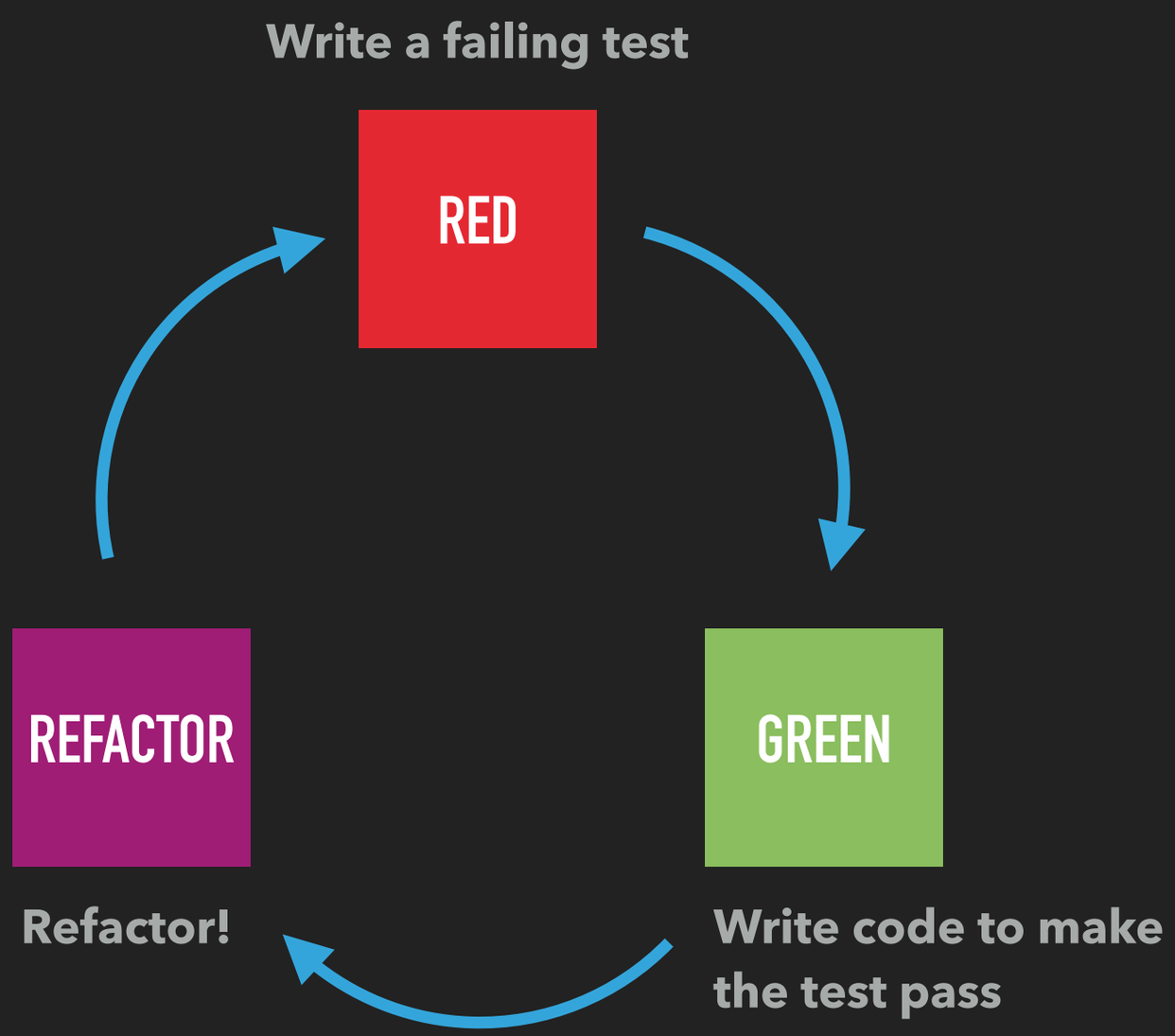
MAKING ASSERTIONS

- There is also an assert function that always fails
    - `XCTAssertFail`

TEST-DRIVEN DEVELOPMENT

- Test-driven development (TDD) is the practice of writing tests *before* implementation code
- The TDD workflow is often described as red-green-refactor

RED-GREEN-REFACTOR



RED-GREEN-REFACTOR

```
23
24      func average(of values: [Double]) -> Double {
25          return 0
26      }
27
28
30      func testAverage() {
31          let values = [2.5, 3.0, 6.5]
32          let avg = average(of: values)
33          XCTAssertEqual(avg, 4.0)
```

Empty implementation

Failing test

XCTAssertEqual failed: ("0.0") is not equal to ("4.0")

RED-GREEN-REFACTOR

```
24      func average(of values: [Double]) -> Double {
25
26          var sum: Double = 0
27          for value in values {
28              sum += value
29          }
30
31          return sum / Double(values.count)
32      }
33
34
        func testAverage() {
36          let values = [2.5, 3.0, 6.5]
37          let avg = average(of: values)
38          XCTAssertEqual(avg, 4.0)
39      }
```

Implementation

Passing test

RED-GREEN-REFACTOR

```
23      func average(of values: [Double]) -> Double {
24          return values.reduce(0, +) / Double(values.count)
25      }
26
27
        func testAverage() {
29          let values = [2.5, 3.0, 6.5]
30          let avg = average(of: values)
31          XCTAssertEqual(avg, 4.0)
32      }
```

Refactored implementation

Passing test

RED-GREEN-REFACTOR

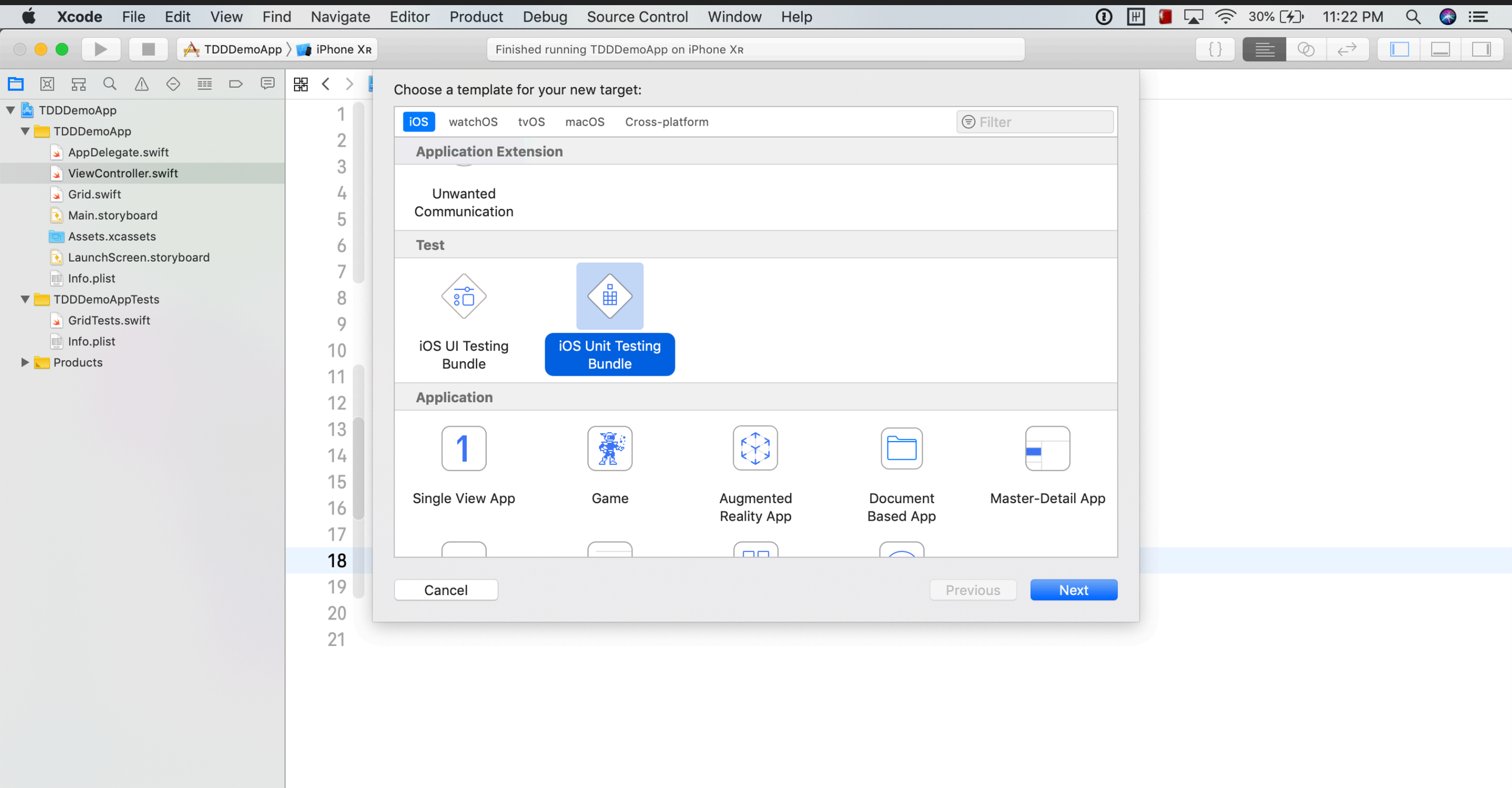
```
23      func average(of values: [Double]) -> Double {
24          return values.reduce(0, +) / Double(values.count)
25      }
26
27
        func testAverage() {
29          let values = [2.5, 3.0, 6.5]
30          let avg = average(of: values)
31          XCTAssertEqual(avg, 4.0)
32      }
33
34
        func testAverage_whenValuesIsEmpty() {
36          let values: [Double] = []
37          let avg = average(of: values)
38          XCTAssertEqual(avg, 0)    XCTAssertEqual failed: ("-nan") is not equal to ("0.0")
39      }
```

Failing test

RED–GREEN–REFACTOR

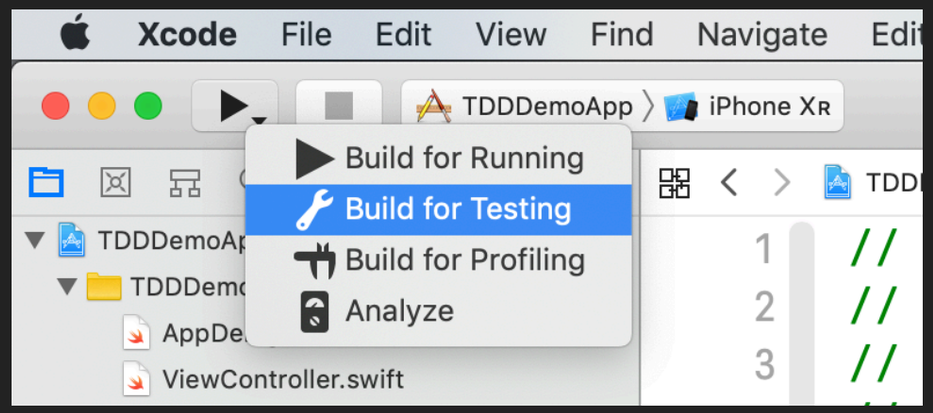
- Why start with a test that fails?

ADDING A TEST TARGET



RUNNING TESTS IN XCODE

- Command+U will run all tests
- Holding down the run button will reveal a test option



- Buttons next to each test class and method let you run individual tests

```
9   import XCTest
10  @testable import TDDDemoApp
11
13  class GridTests: XCTestCase {
14
14      private let grid = Grid()
15
17      func testPieceAtIndex_whenSquareIsOccupied() {
17          grid.add(piece: .x, at: 0)
18          XCTAssertEqual(grid.piece(at: 0), .x)
19      }
```

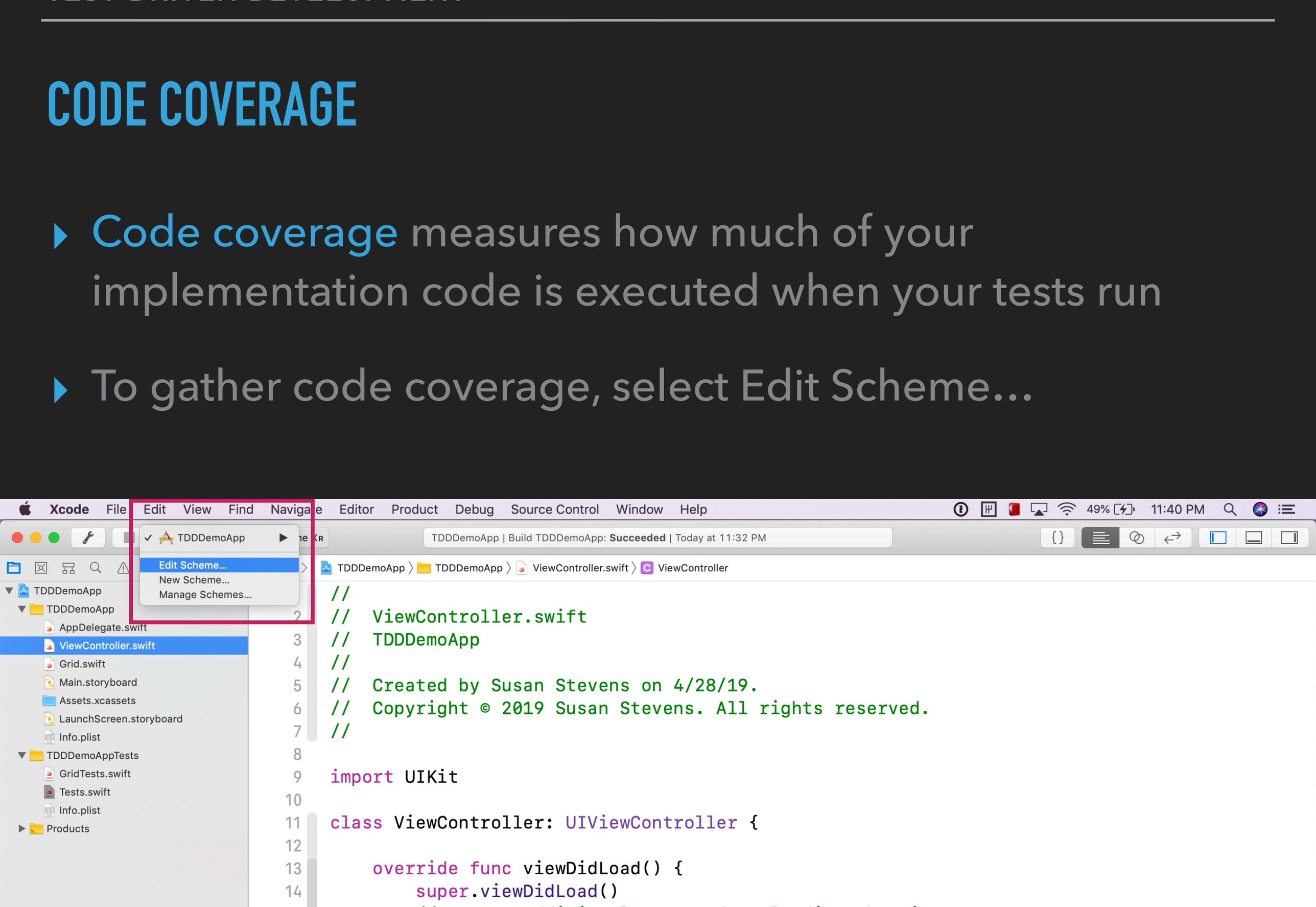
ANATOMY OF A TEST CLASS

```
8
9   import XCTest
10  @testable import TDDDemoApp
11
13  class Tests: XCTestCase {
14
15      override func setUp() {
            // Put setup code here. This method is called before the invocation of each test
                method in the class.
16      }
17
18      override func tearDown() {
19          // Put teardown code here. This method is called after the invocation of each test
                method in the class.
20      }
21
23      func testExample() {
24          // This is an example of a functional test case.
            // Use XCTAssert and related functions to verify your tests produce the correct
                results.
25      }
26
27      func notATestExample() {
28          XCTAssertTrue(false)
29      }
```

1. Import XCTest framework and your application code
2. Subclass XCTestCase
3. Use setUp method for common set up code
4. Use tearDown to clean up after each test runs
5. Tests must start with the word “test”
6. Xcode will *not* see this as a test

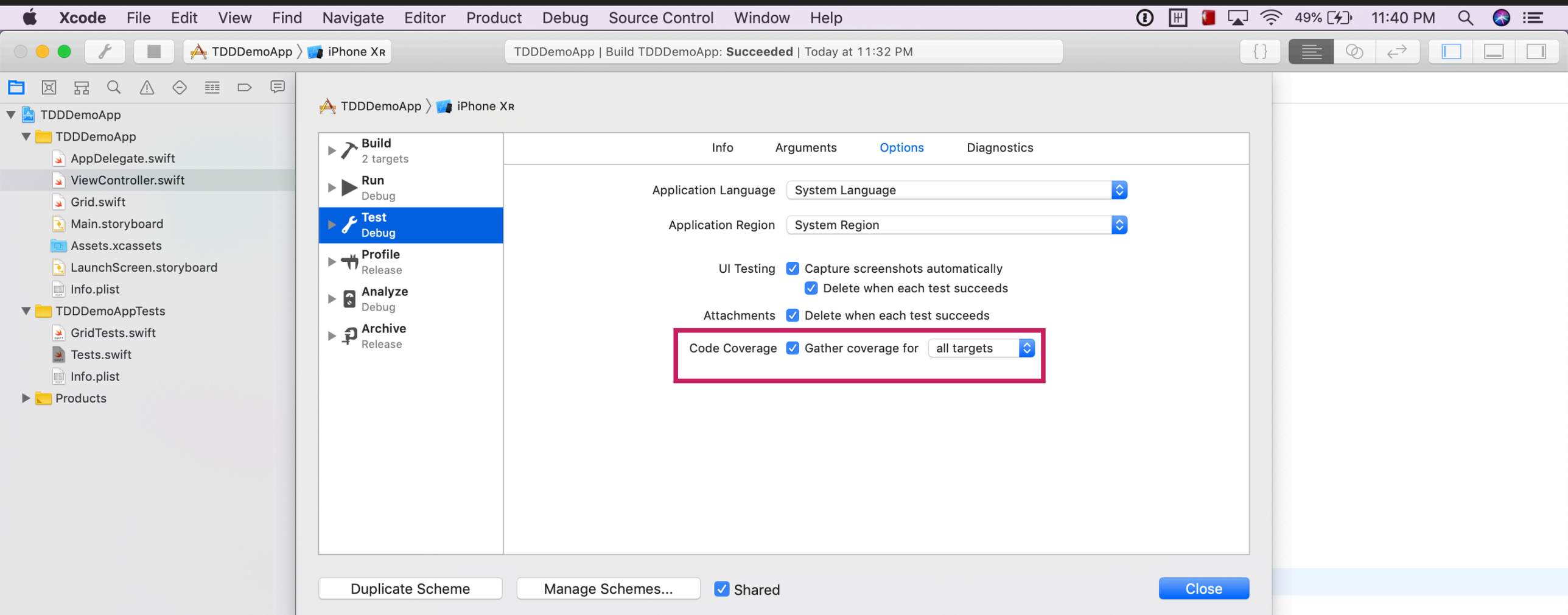
CODE COVERAGE

- Code coverage measures how much of your implementation code is executed when your tests run
- To gather code coverage, select Edit Scheme…



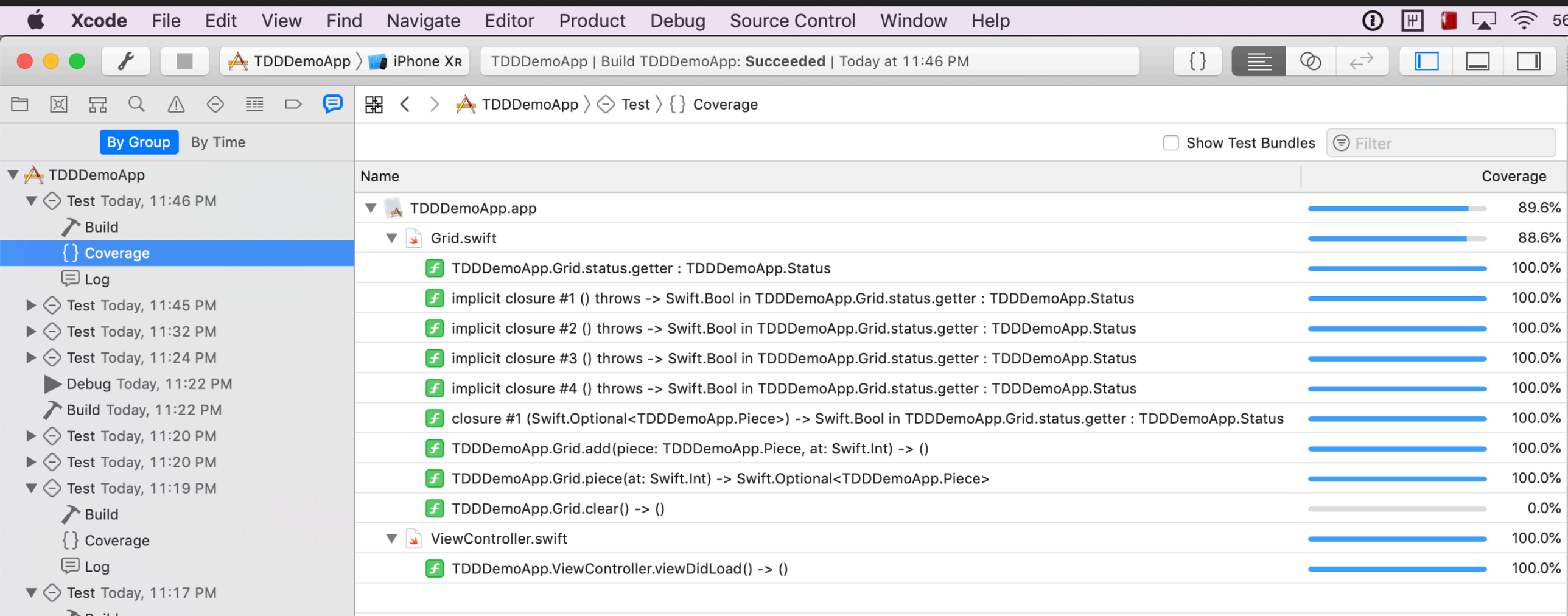
CODE COVERAGE

- Under Test > Options, check “Gather coverage for all targets”



CODE COVERAGE

- After running your tests, you'll be able to see a breakdown of how well your code is covered by tests



REMINDERS

FINAL PROJECTS

REMINDERS

- Final project presentations will be on Wednesday, March 18
- Projects are due on March 18 @ 11:59 PM
- Review the final project requirements on the course website: uchicago.mobi/final-projects